CS 5010: Programming Design Paradigms
Fall 2019

Lecture 7: Data Structures and Algorithms

Acknowledgement: lecture notes inspired by course material prepared by
UW faculty members Z. Fung and H. Perkins.

Tamara Bonaci
t.bonaci@northeastern.edu

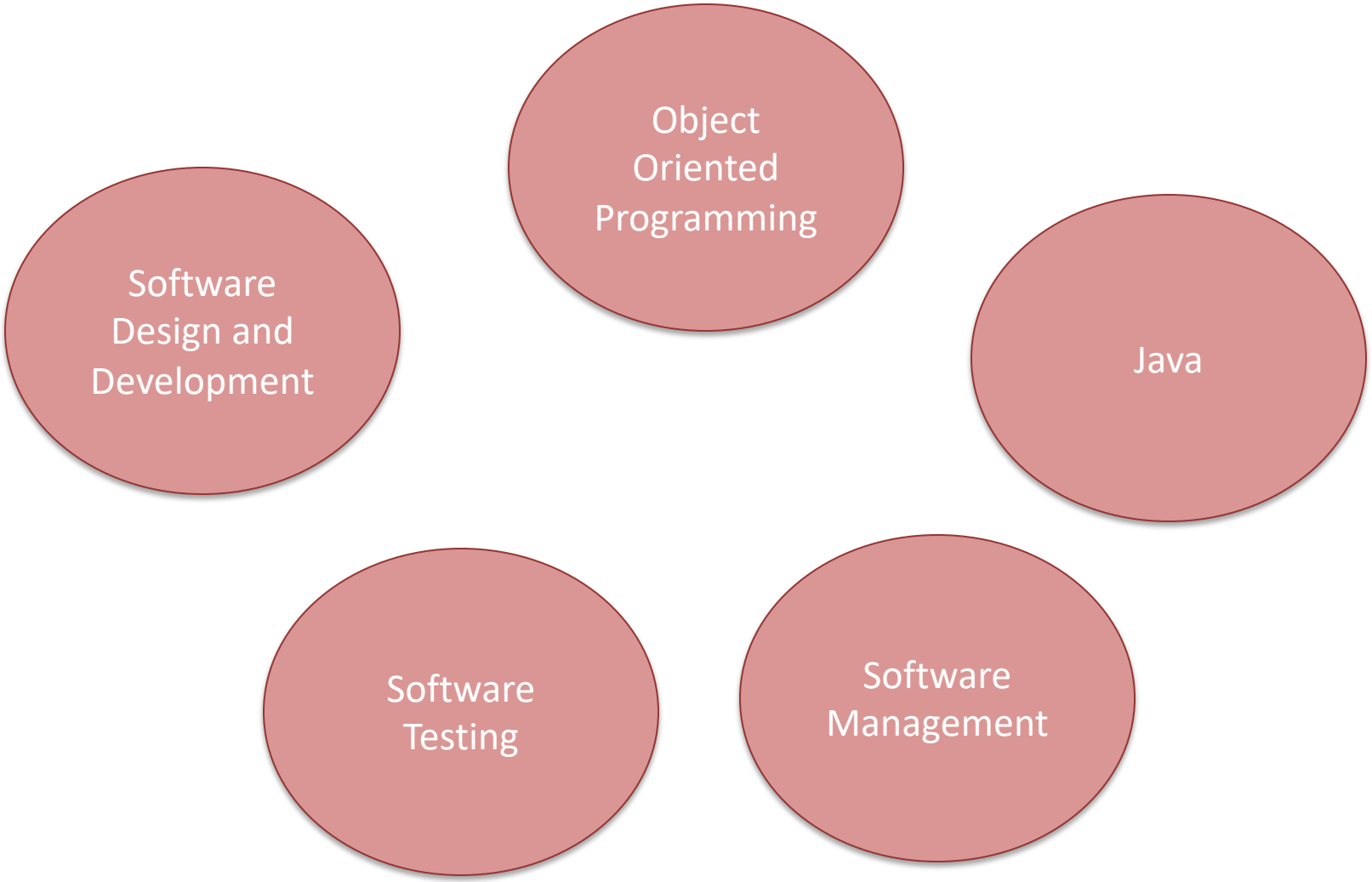
Administrivia

- **Assignment 4** due on Monday, October 21 by 6pm
- **Codewalk 4** next week on Tue and Wed, Oct 22 and 23
- **Lab 3** – nothing to submit, but please do it before A4 deadline
- (new work process --> we're not using Bottlenose anymore)

Agenda – Algorithms and Data Structures 1

- Data collections
  - Iterating over data collections
    - Interface Iterable
    - Iterator
  - Ordering of objects
    - Interface Comparable
    - Interface Comparator
- List ADT
  - Doubly-linked List
  - Inner and nested classes
  - Algorithm: Recursion
  - Recursive data collections
- Stack ADT
- Queue ADT

Course so Far…



Objects and Classes

- Object – an entity consisting of states and behavior
  - States stored in variables/fields
  - Behavior represented through methods

- Class – template/blueprint describing the states and the behavior that an object of that type supports

Interface vs. Abstract Class vs. Concrete Class?

[Pictures credit: http://www.shakespearesglobe.com/https://www.vexels.com/https://www.amazon.com/Concrete-Leonard-Koren/dp/0714863548]

Abstract Class vs. Abstract Data Type?

Vs.

[Pictures credit: https://www.vexels.com/, https://getyouralarm.com]

- Abstract class – contains one or more abstract methods
- ADT – a high-level model of a data type, where the data type is defined from a point of view of the user (focus on operations (behavior), not on implementation)

Review: Abstract Data Type

- Abstract Data Type (ADT) - model that describes data by specifying the operations that we can perform on them
- Clients care about the ADT → we need to capture the clients expectations in terms of the operations on the ADT
- For each operation, we need to describe:
  - The expected inputs, and any conditions that need to hold for our inputs and/or our ADT
  - The expected outputs and any conditions that need to hold for our output and/or our ADT
  - Invariants about our ADT

Object Oriented Design Principles

- Encapsulation
- Abstraction
- Information hiding
- Polymorphism
- Inheritance

Review: Polymorphism

Polymorphism – the ability of one instance to be viewed/used as different types (the ability to take many shapes/forms/views)

Three Types of Polymorphism

Polymorphism – the ability to define different classes and methods as having the same name but taking different data types

Three types of polymorphism:

1. Subtype polymorphism
2. Ad hoc polymorphism
3. Parametric polymorphism

Review: Subtype Polymorphism

- Subtypes are substitutable for supertypes
  - Instance of subtypes won't surprise a client by failing to satisfy the supertype's specification
  - Instance of subtype won't surprise a client by having more expectations than the supertypes' specifications

Review: Compile Time and Run Time Types

```
Person emily = new Person();
Singer adele = new Singer();
Person flora = new Singer();
```

- Static (compile time) type – the declared type of a reference variable. Used by a compiler to check syntax
- Dynamic (run time) type – the type of an object that the reference variable currently refers to (it can change as the program execution progresses)

Review: Ad Hoc Polymorphism

- Overloading allows us to create methods that share the same method name but differ in their signature

- Ad hoc polymorphism – another name for function and operator overloading

- Ad hoc polymorphism – a type of polymorphism where a polymorphic functions can be applied to arguments of different types
  - Polymorphic (overloaded) function can denote a number of distinct and potentially heterogeneous implementations, depending on the type of argument(s) to which it is applied

Parametric Polymorphism

- Parametric polymorphism - ability for a function or type to be written in such a way that it handles values identically without depending on knowledge of their types
  - Such a function or type is called a generic function or data type

- Motivation - parametric polymorphism allows us to write flexible, general code without sacrificing type safety
  - Most commonly used in Java with collections

Algorithms and Data Structures 1

ASIDE: BASIC JAVA I/O

I/O Streams

- Concept of an I/O stream – a communication channel ("pipe") between a source and a destination that allows us to create a flow of data

- I/O Stream has:
  - An input source (a file, another program, device)
  - An output destination (file, another program, device)
  - The kind of data streamed (bytes, ints, objects)

How to Do I/O

```
import java.io.*;
```

- Open the stream
- Use the stream (read, write, or both)
- Close the stream

open
use
close

Opening a Stream

- Problem:
  - There exists some external data that we want to get, or
  - We want to put data somewhere outside your program
- Solution: open a steam
  - When we open a stream, we are making a connection to that external place
  - Once the connection is made, we can forget about the external place, and just use the stream

open
use
close

Example of Opening a Stream

- `FileReader` - used to connect to a file that will be used for input:

```
FileReader fileReader = new FileReader(fileName);
```

  - Filename - specifies where to find the (external) file (Note: after instatiating `FileReader` object, we never use `fileName` again; instead, we use `fileReader` object

open
use
close

Using a Stream

- Using a stream means doing input from it, or output to it
- Some streams can be used only for input, others only for output, still others for both
- But it is not usually that simple - we need to manipulate the data in some way as it comes in, or goes out

open
use
close

Example of Using a Stream

```
int ch;
ch = fileReader.read( );
```

- The `fileReader.read()` method reads one character, and returns it as an integer, or -1 if there are no more characters to read
- The meaning of the integer depends on the file encoding (ASCII, Unicode, other)

open
use
close

Manipulating the Input Data

- Reading characters as integers is not usually what we want to do
- A `BufferedReader` will convert integers to characters; it can also read whole lines
- The constructor for `BufferedReader` takes a `FileReader` parameter:

```
BufferedReader bufferedReader = new
BufferedReader(fileReader);
```

open
use
close

Reading Lines

```
String s;
s = bufferedReader.readLine( );
```

- A `BufferedReader` will return null if there is nothing more to read

BufferedReader

- Reads text from a character-input stream, buffering characters to provide efficient reading of characters, arrays, and lines
- Buffer size may be specified, or the default size may be used (the default is large enough for most purposes)

- Why use BufferReader?
  - In general, each read request causes a corresponding read request to be made of the underlying character or byte stream
  - These operations may be costly → each invocation of `read()` or `readLine()` in a `FileReader` or `InputFileReader` causes bytes to be read from the file, converted into characters, and then returned
  - `BufferReader` increases efficiency by buffering the input from the specified file

open
use
close

Closing a Stream

- A stream is an expensive resource!
- There is a limit on the number of streams that you can have open at one time
- You should not have more than one stream open on the same file
- You should close a stream before you can open it again
- Always close your streams!

Algorithms and Data Structures 1

DATA COLLECTIONS

Data Collections?

Collection of chewed gums

Collection of pens

Collection of cassette tapes

Collection of old radios

What is a data collection?

Shoes collection

Star wars collection

Cars collection

[Pictures credit: http://www.smosh.com/smosh-pit/articles/19-epic-collections-strange-things]

Data Collections?

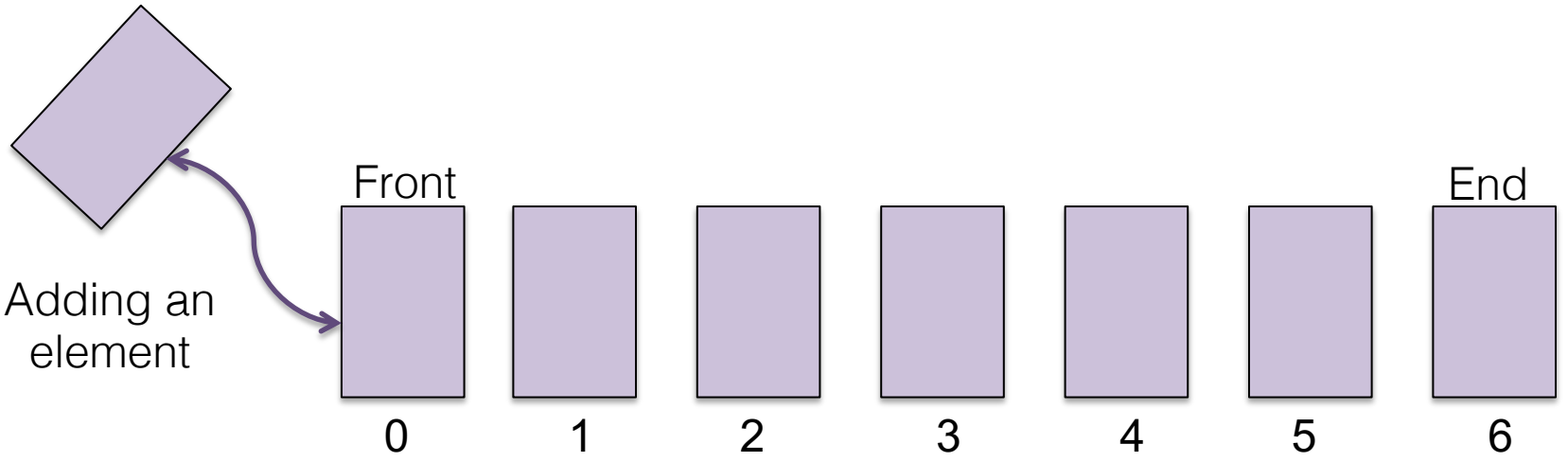
- Data collection - an object used to store data (think *data structures*)
  - Stored objects called elements
  - Some typically operations:
    - `add()`
    - `remove()`
    - `clear()`
    - `size()`
    - `contains()`
- Some examples: ArrayList, LinkedList, Stack, Queue, Maps, Sets, Trees

Why do we need different data collections?

Example: ArrayList vs. LinkedList

- **List** - a collection of elements with 0-based **indexes**
  - Elements can be added to the front, back, or in the middle
  - Can be implemented as an ArrayList or as a LinkedList

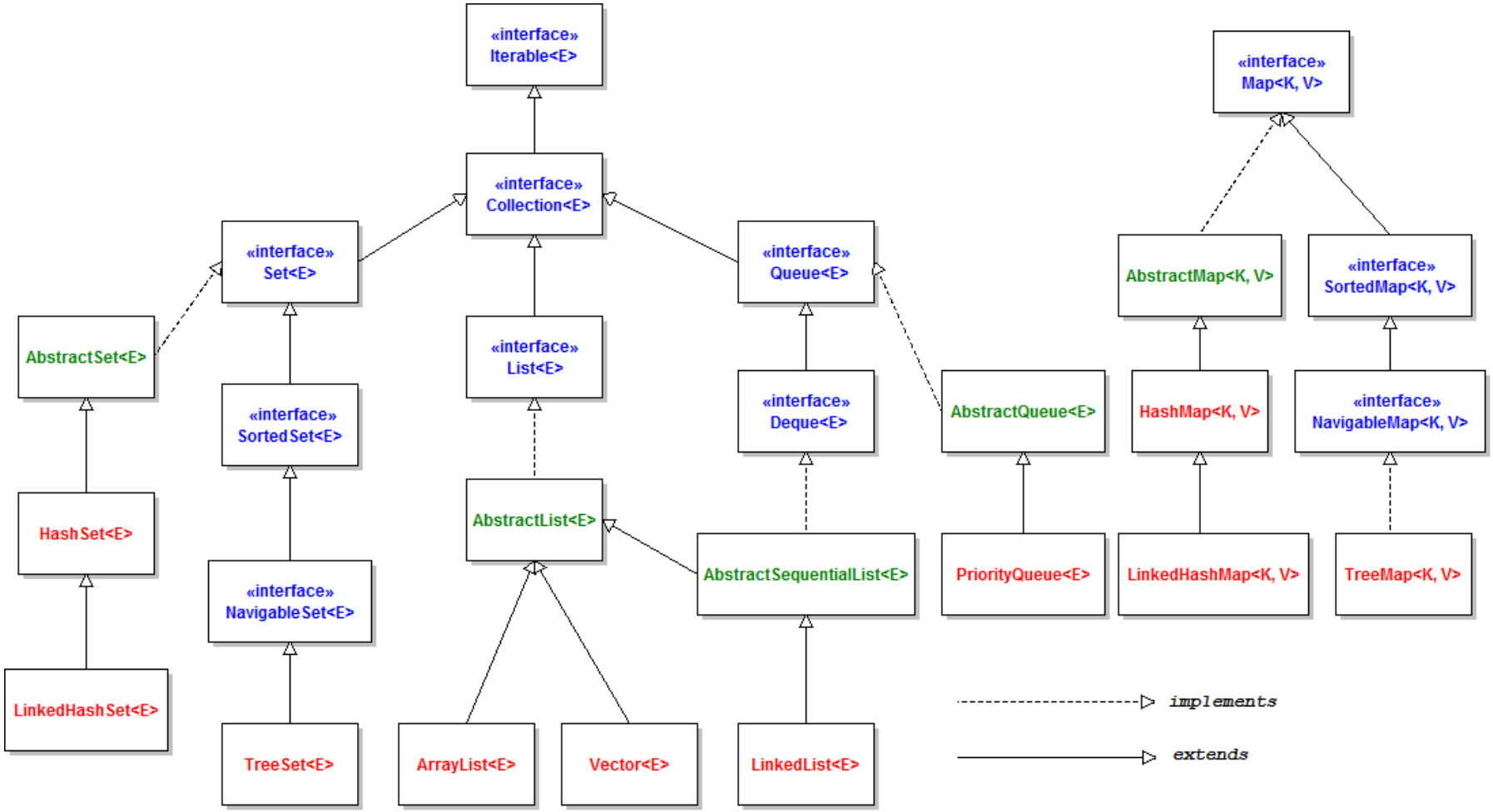
  - What is the complexity of adding an element to the front of an:
    - ArrayList?
    - LinkedList?



Algorithms and Data Structures 1

JAVA COLLECTIONS FRAMEWORK

Java Collections API



[Pictures credit: http://www.codejava.net]

Java Collections Framework

- Under the Java Collections Framework, we have:
  - Interfaces that define the behavior of various data collections
  - Abstract classes that implement the interface(s) of the collection framework, that can then be extended to created a specialized data collection
  - Concrete classes that provide a general-purpose implementation of the interface(s)

Java Collections Framework

- Goals of the Java Collections Framework:
  1. Reduce programming effort by providing the most common data structures
  2. Provide a set of types that are easy to use and extend
  3. Provide flexibility through defining a standard set of interfaces for collections to implement
  4. Improve program quality through the use and reuse of tested implementations of common data structures

Java Collections Framework

- Part of the `java.util` package

- Interface `Collection<E>`:
  - Root interface in the collection hierarchy
  - Extended by four interfaces:
    - `List<E>`
    - `Set<E>`
    - `Queue<E>`
    - `Map<K,V>`
  - Extends interface `Iterable<T>`

Interfaces Iterable<T>

- Super-interface for interface `Collection<T>`
- Implementing interface `Iterable<T>` allows an object to be traversed using the for each loop
- Every object that implements `Iterable<T>` must provide a method `Iterator iterator()`

Modifier and Type	Method and Description
`default void`	**`forEach(Consumer<? super T> action)`** Performs the given action for each element of the `Iterable` until all elements have been processed or the action throws an exception.
`Iterator<T>`	**`iterator()`** Returns an iterator over elements of type T.
`default Spliterator<T>`	**`spliterator()`** Creates a **`Spliterator`** over the elements described by this `Iterable`.

Interfaces `Iterable<T>` and forEach Loop

- Super-interface for interface `Collection<T>`
- Implementing interface `Iterable<T>` allows an object to be traversed using the for each loop
- Every object that implements `Iterable<T>` must provide a method `Iterator iterator()`

- `ForEach loop:`
  default void `forEach(Consumer<? super T> action)`

- Performs the action for each element of the interface `Iterable`, until:
  - All elements have been processed or
  - The action throws an exception
- Actions are performed in the order of iteration (unless otherwise specified by the implementing class)
- Exceptions thrown by the action are relayed to the caller

Interfaces `Iterable<T>` and `Iterator<E>`

- Super-interface for interface `Collection<T>`
- Implementing interface `Iterable<T>` allows an object to be traversed using the for each loop
- Every object that implements `Iterable<T>` must provide a method `Iterator iterator()`

- Interface `Iterator` – an iterator over a collection

```
public interface Iterator<E> {
   boolean hasNext();
   E next();
   void remove():
}
```

- Iterator `remove()` method – removes the last item returned by method `next()`

Direct Use of an `Iterator<E>`

- Interface `Iterator` – an iterator over a collection

```
public interface Iterator<E> {
   boolean hasNext();
   E next();
   void remove():
}
```

- Careful when an iterator is used directly (not via a for each loop)→ if you make any structural changes to a collection being iterated (add, remove, clear), the iterator is no longer valid (`ConcurrentModificationException` thrown)

Algorithms and Data Structures 1

NATURAL ORDERING OF OBJECTS

Comparing Objects

- Problem: How do we compare `String` in some list of `Strings`?
  - Operators like < and > do not work with `String` objects
  - But we do think of `Strings` as having an alphabetical ordering
- Natural ordering: Rules governing the relative placement of all values of a given type
- Comparison function: Code that, when given two values A and B of a given type, decides their relative ordering:

  - A < B, A == B, A > B

The `compareTo` method

- A standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method
  - Example: in the `String` class, there is a method:

```
public int compareTo(String other)
```

- A call of A.`compareTo(B)` will return:

  a value < 0 if A comes "before" B in the ordering,

  a value > 0 if A comes "after" B in the ordering,

  0 if A and B are considered "equal" in the ordering.

Using compareTo

- `compareTo` can be used as a test in an if statement

```
String a = "alice";
String b = "bob";
if (a.compareTo(b) < 0) {   // true
    ...
}
```

Primitives	Objects
`if (a < b) { ...`	`if (a.compareTo(b) < 0) { ...`
`if (a <= b) { ...`	`if (a.compareTo(b) <= 0) { ...`
`if (a == b) { ...`	`if (a.compareTo(b) == 0) { ...`
`if (a != b) { ...`	`if (a.compareTo(b) != 0) { ...`
`if (a >= b) { ...`	`if (a.compareTo(b) >= 0) { ...`
`if (a > b) { ...`	`if (a.compareTo(b) > 0) { ...`

`compareTo()` and Java Collections

- We can use an array or list of `Strings` with Java's included binary search method because it calls `compareTo` internally

```
String[] a = {"al", "bob", "cari", "dan", "mike"};
int index = Arrays.binarySearch(a, "dan");  // 3
```

- Java's `TreeSet/Map` use compareTo internally for ordering

```
Set<String> set = new TreeSet<String>();
for (String s : a) {
    set.add(s);
}
System.out.println(s);
// [al, bob, cari, dan, mike]
```

Ordering Our Own Types

- Problem: we cannot binary search or make a `TreeSet/Map` of arbitrary types, because Java doesn't know how to order the elements
- Example: the program compiles but crashes when we run it

```
Set<HtmlTag> tags = new TreeSet<HtmlTag>();
tags.add(new HtmlTag("body", true));
tags.add(new HtmlTag("b", false));
…
Exception in thread "main"
  java.lang.ClassCastException
        at
  java.util.TreeSet.add(TreeSet.java:238)
```

Comparable Template

```
public class name implements Comparable<name> {

    ...

    public int compareTo(name other) {
        ...
    }
}
```

Comparable Example

```
public class Point implements Comparable<Point> {
    private int x;
    private int y;
    ...


    // sort by x and break ties by y
    public int compareTo(Point other) {
        if (x < other.x) {
            return -1;
        } else if (x > other.x) {
            return 1;
        } else if (y < other.y) {
            return -1;    // same x, smaller y
        } else if (y > other.y) {
            return 1;     // same x, larger y
        } else {
            return 0;     // same x and same y
        }
    }
}
```

Interface Comparator

- Problem: we may want to be able to order instance of some classes by more then one property (for example, by first name and by last name). What can we do?
- Solution: use interface `Comparator<T>` - a comparison function, which imposes a total ordering on some collection of objects
  - Can be passed to a sort method, to allow precise control over the sort order
  - Can also be used to control the order of certain data structures (tree sets and tree maps)
  - Can also be used provide an ordering for collections of objects that don't have a natural ordering

Method `compare(T o1, T o2)`

- `int compare(T o1, T o2)` - compares its two arguments for order and returns:
  - A negative integer if the first argument is less than the second
  - Zero, if the first argument is equal to the second
  - A positive integer if the first argument is greater than the second
- The implementor must also ensure that the relation is transitive:

```
((compare(x, y) > 0) && (compare(y, z) >
0)) implies compare(x, z) > 0
```

- Not strictly required, but good rule to follows: `compare()` method should be consistent with `equals()` method:

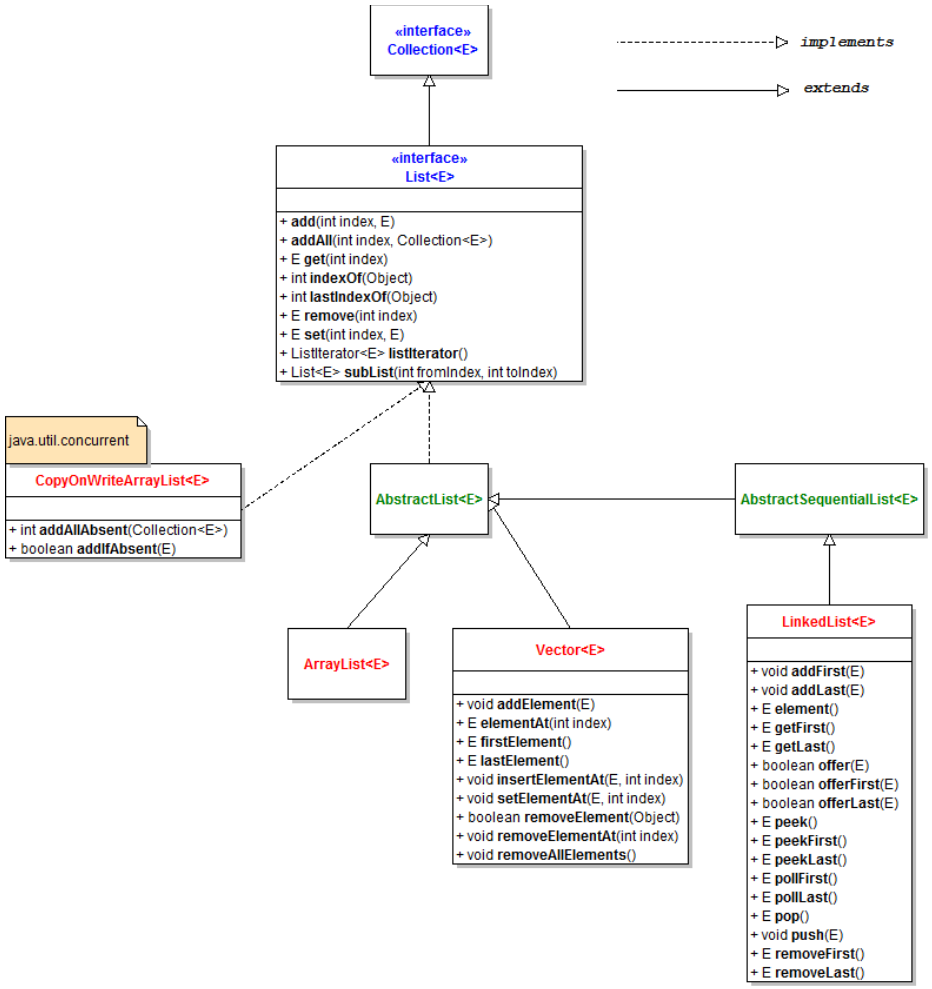
```
(compare(x, y)==0) == (x.equals(y))
```

(Any comparator that violates this condition should clearly indicate this fact. The recommended language is "`Note: this comparator imposes orderings that are inconsistent with equals.`") (don't do this!)

Algorithms and Data Structures 1

LIST ADT

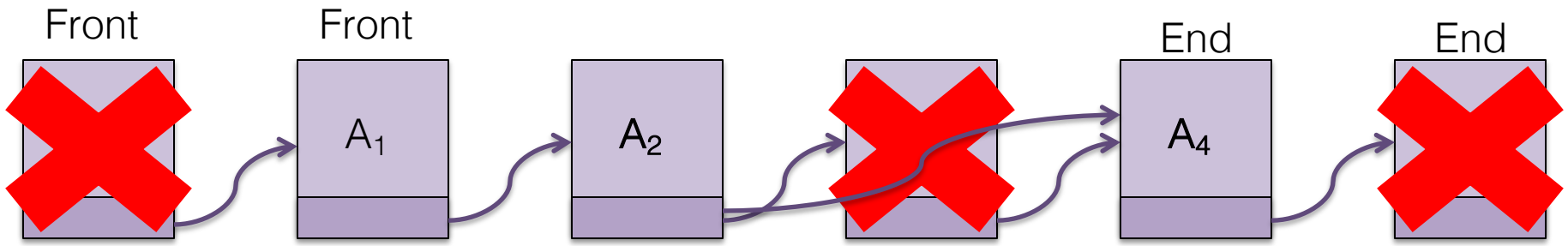
Java List API



[Pictures credit: http://www.codejava.net]

- `List<E>` - the base interface
- Abstract subclasses:
  - `AbstractList<E>`
  - `AbstractSequentialList<E>`
- Concrete classes:
  - `ArrayList<E>`
  - `LinkedList<E>`
  - `Vector<E>` (legacy collection)
  - `CopyOnWriteArrayList<E>` (class under java.util.concurrent package)
- Main methods:
  - `E get(int index);`
  - `E set(int index, E newValue);`
  - `Void add(int index, E x);`
  - `Void remove(int index);`
  - `ListIterator<E> listIterator();`

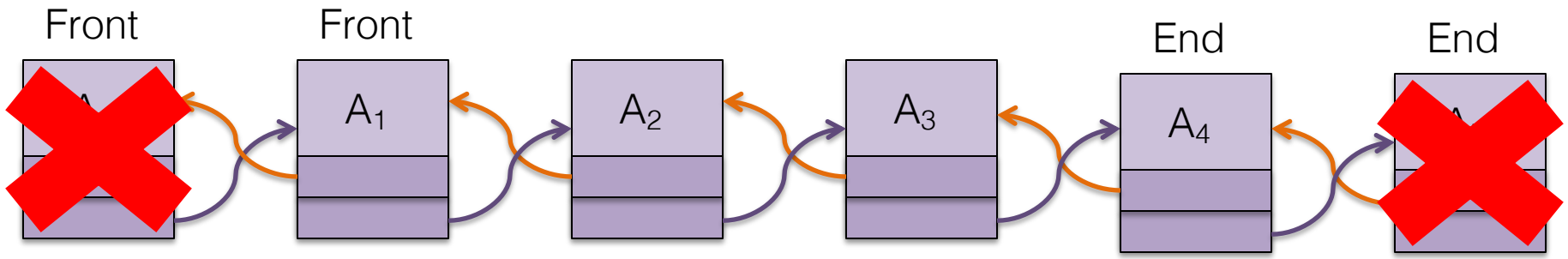
Example: Removing Elements from a LinkedList



- Remove the first element of the list
- Remove element A_3 from the list
- Remove the last element in the list

- What's the tricky part about removing elements A_3 and A_5?
- Having to find their predecessors (elements A_2 and A_4) and updating their link to last node
- Idea: every node maintains the link to its previous and its next node → doubly linked list

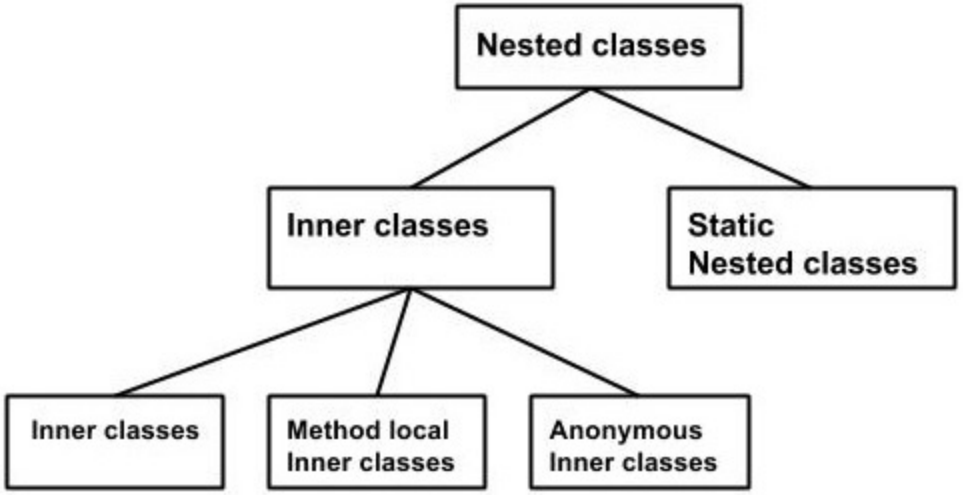
Doubly LinkedList



- Removing the first element of the list
- Removing the last element of the list

Implementation of a Doubly LinkedList

- Review – inner and nested classes:



[Pictures credit: https://www.tutorialspoint.com/java/images/inner_classes.jpg]

- Java allows us to define a class *within* another class – a nested class
  - Static nested classes
  - Non-static nested classes or inner classes – have access to other members of the enclosing class, even if they are declared private

Nested Classes

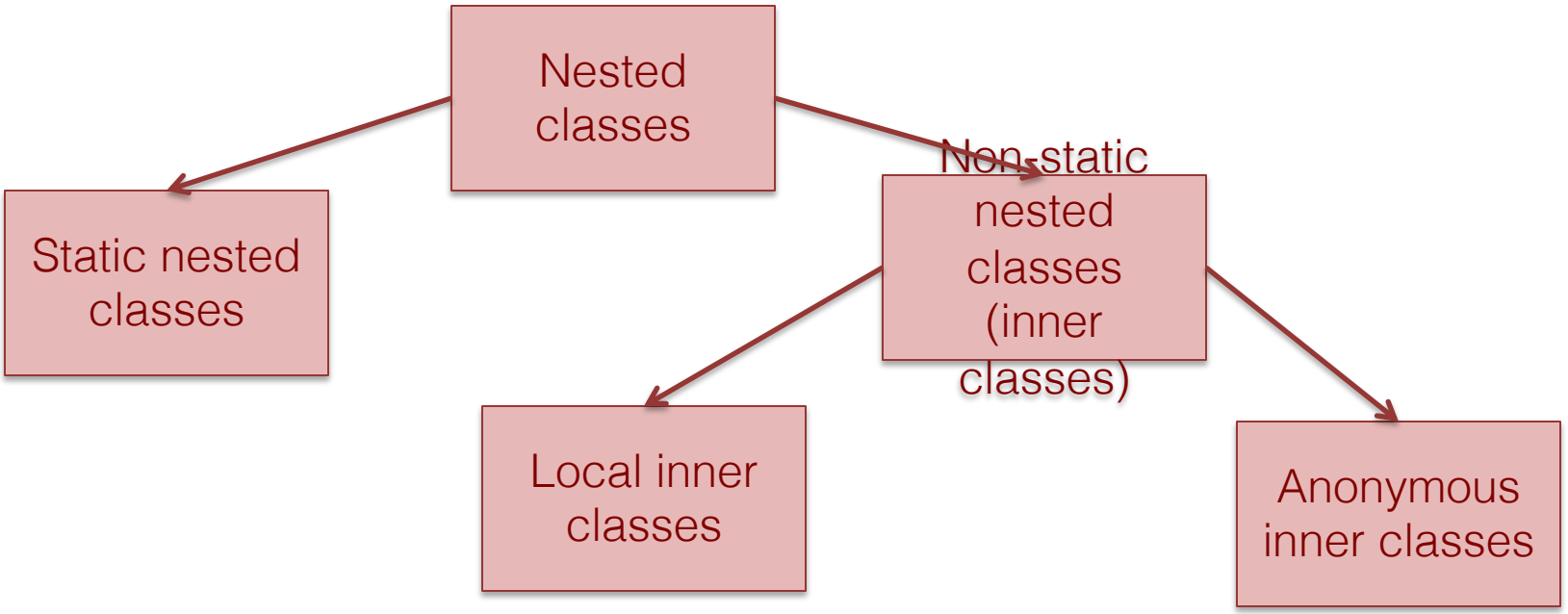
- Nested class – a class defined within some other class

- Some nested class features:
  - The scope of a nested class is bounded by the scope of its outer class (i.e., the inner class does not exist independently of its outer class)
    - Class example: node and tree
  - A nested is a member of its outer class
    - Can be declared private, public, protected
    - Can have access to all members (including private) of its outer class
    - Outer class does not have access to the members of the nested class

Nested Classes

- Nested class – a class defined within some other class

- Nested classes can be:
  - Static classes
  - Non-static (inner classes)



Static Nested Classes

- Behave the same way as top-level classes

```
class A {
//code for A
   static class B {
   //code for B
   }
}
```

- To access a static nested class, we need to use the name of the outer class:

```
   A.B b = new A.B()
```

Inner Classes

- Object of inner classes exist within an instance of the outer class

```
class X {
//code for X
   Class Y {
   //code for Y
   }
}
```

- To access an inner nested class, we need to do so through an instance of the outer class:

```
X x = new X();
   X.Y y = x.new Y();
```

Difference Between Static and Inner Classes

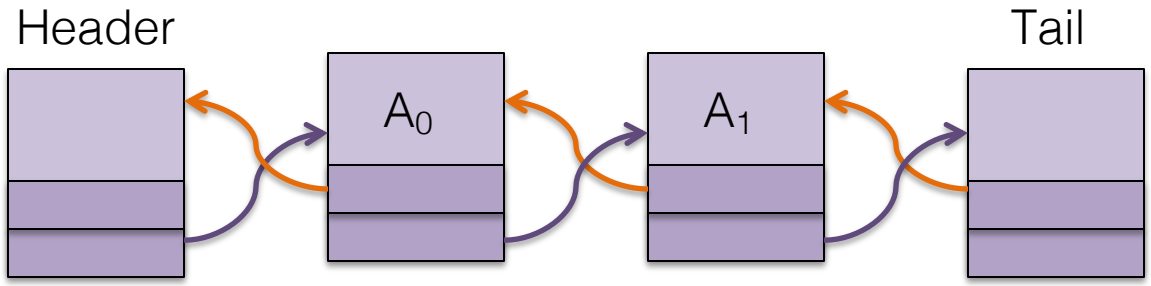
- Static nested classes do not have a direct access to the non-static members of the outer class (non-static variables and methods)
  - Static class must access the non-static members of its enclosing class through an object

- Inner classes have access to all members (static and non-static, including private) of its outer class, and may refer to them directly
  - Used more frequently

Implementation of a Doubly LinkedList

- Doubly linked list – need to provide and maintain links to both ends of the list
- Implemented classes:
  - Class `MyLinkedList`
  - Class `Node` – private nested class, contains the data and links to previous and next nodes
  - Class `LinkedListIterator` – private inner class, implementing the interface `Iterator`

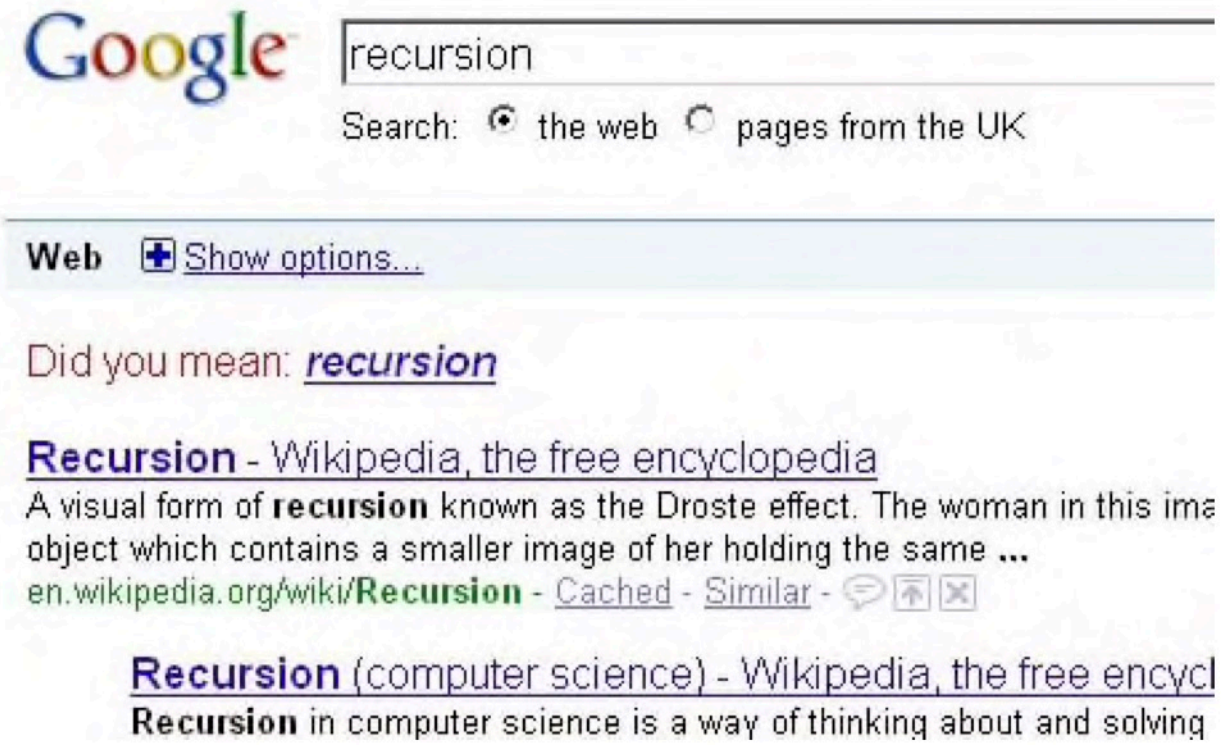
- Sentinel nodes:
  - Header and tail nodes, used to logically represent the beginning and the end markers



Algorithms and Data Structures 1

LIST ADT AND RECURSIVE DATA COLLECTIONS

Recursion



[Pictures credit: http://www.telegraph.co.uk/technology/google/6201814/Google-easter-eggs-15-best-hidden-jokes.html]

Recursion

- Recursion – an operation defined in terms of itself
  - Solving a problem recursively means solving smaller occurrences of the same problem
- Recursive programming – an object consist of methods that call themselves to solve some problem
- Can you think of some examples of recursions and recursive programs?

Recursive Algorithm

- Every recursive algorithm consists of:
  - Base case – **at least one** simple occurrence of the problem that can be answered directly
  - Recursive case - more complex occurrence that cannot be directly answered, but can be described in terms of smaller occurrences of the same problem

- A crucial part of recursive programming is identifying these cases
- What were base cases for our Fibonacci problem in Assignment 1?
- Why are we now talking about recursion???

Recursive Data Structures

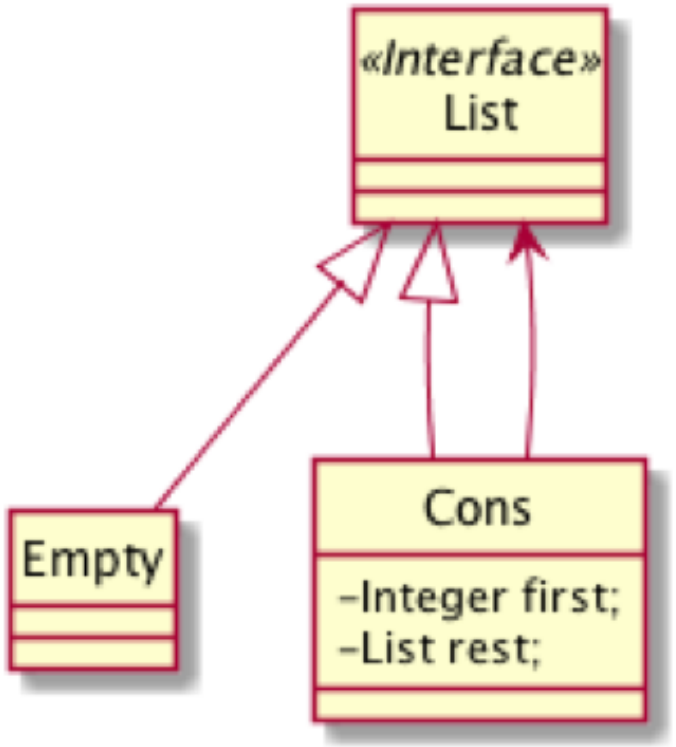
- Recursive data structure - a data structure partially composed of smaller or simpler instances of the same data structure

- Is linked list a recursive data structure?
- Let's see - a linked list is either
  - Null (base case)
  - A node whose next field references a list

Lists as a Recursive Data Collection

- List – an ordered collection (also known as a sequence)

- A linked list is either:
  - Null (base case)
  - A node whose next field references a list

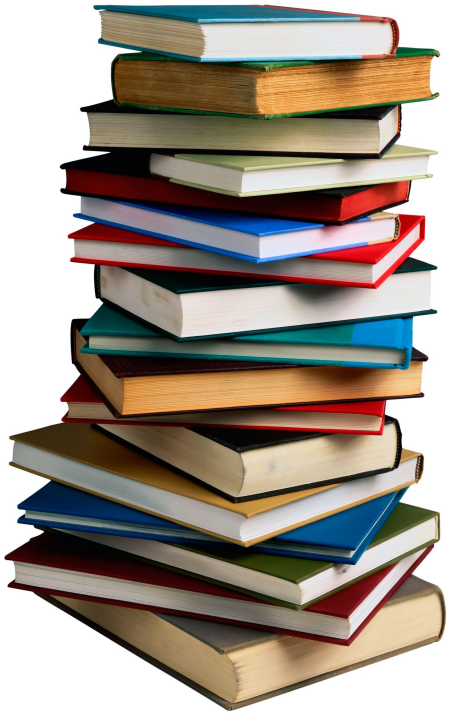


Algorithms and Data Structures 1

STACK ADT

Stacks

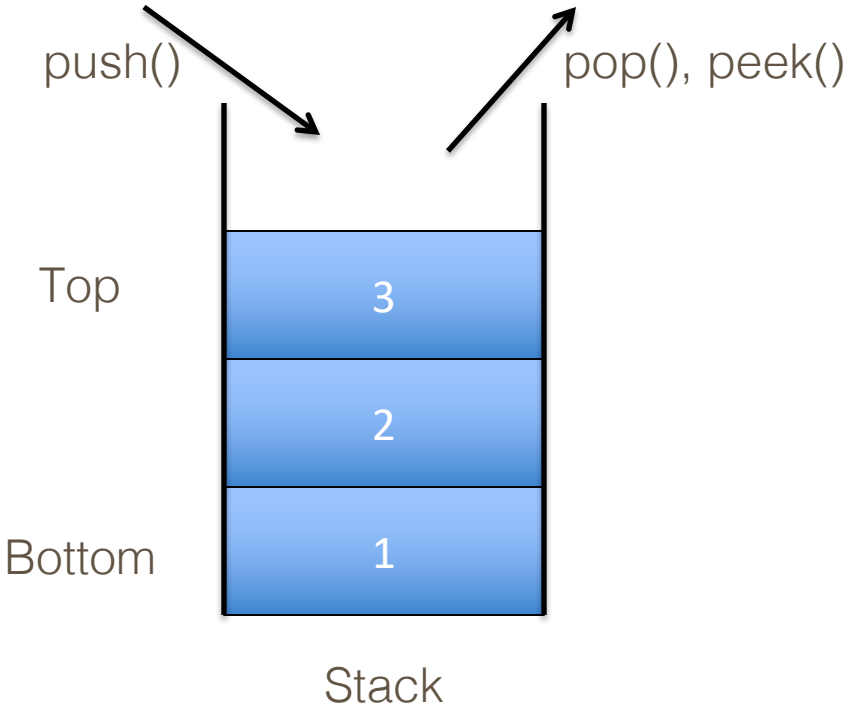
- Some of my favorite stacks:



[Pictures credit: https://rukminim1.flixcart.com/image/1408/1408/stacking-toy, http://battellemedia.com/wp-content/uploads/2014/08/National-Pancake-Day-at-IHOP.jpg, http://all4desktop.com/data_images/original/4245681-book.jpg]

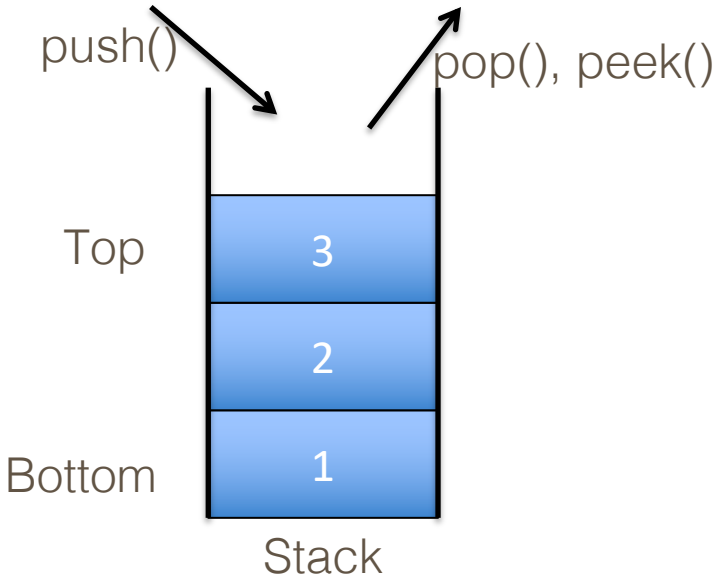
What is a Stack?

- Stack – a data collection that retrieves elements in the LIFO order (last-in-first-out)



- Is there another way to think about a stack?

What is a Stack?

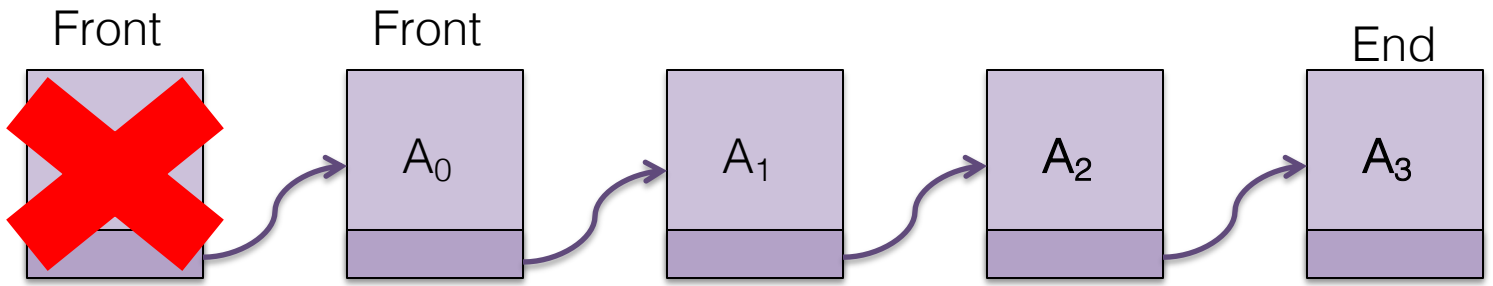


- Is there another way to think about a stack?
- Stack – a constrained data collection where clients are limited to use only limited optimized methods (`pop, push, peek`)
- Stack – a list with restriction that insertions and deletions can be performed in only one position, the end of the list, called the top

Implementations of a Stack

- Stack – a list with restriction that insertions and deletions can be performed in only one position, the end of the list, called the top

- Since a stack is a list, any list implementation will do:
  - Example: linked list implementation
    - `push()`
    - `peek()/top()`
    - `pop()`



Java Class `Stack`

`Stack <E> ()`	Object constructor – constructs a new stack with elements of type E
`push(value)`	Places given value on top of the stack
`pop()`	Removes top value from the stack, and returns it. Throws `EmptyStackException` if the stack is empty.
`peek()`	Returns top value from the stack without removing it. Throws EmptyStackException if the stack is empty.
`size()`	Returns the number of elements on the stack.
`isEmpty()`	Returns true if the stack is empty.

Example:

```
Stack<String> s = new Stack<String>();
s.push("Hello");
s.push("PDP");
S.push("Fall 2019"); //bottom ["Hello", "PDP", "Fall 2019"] top
System.out.println(s.pop()); //Fall 2019
```

Applications of a Stack

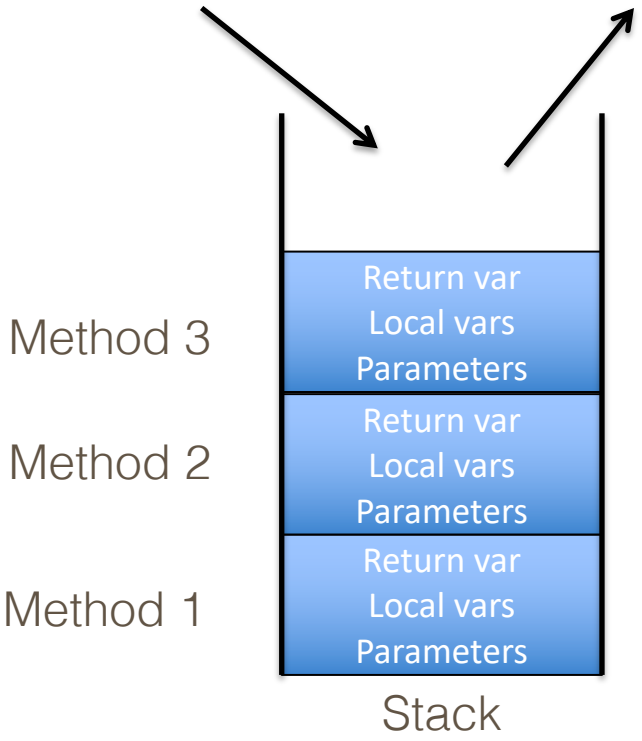
- Programming languages and compilers:
  - Method calls *(call=push, return=pop)*
  - Compilers (parsers)

- Matching up related pairs of things:
  - Find out whether a string is a palindrome
  - Examine a file to see if its braces { } match
  - Convert "infix" expressions to pre/postfix

- Sophisticated algorithms:
  - Searching through a maze with "backtracking"

Example: Methods Call

- In some system, whenever there is a method call, some information about the current state of the system needs to be stored before the control is transferred to a new method:
  - Parameters
  - Local variables
  - Return address



Example: Postfix Expressions

- Suppose you are using a calculator to compute the total cost of your groceries
  - Add the costs of individual items
  - Multiply by 1.1 to account for local sales tax

- The natural way to do this with a calculator:

  5.5 + 4.5 + 7 + 8*1.1

- What is the expected result?
  - 27.5 (expected value)
  - 25.8

- That depends on how "smart" is your calculator!

Example: Postfix Expressions

- The natural way to do this with a calculator:

  5.5 + 4.5 + 7 + 8*1.1

- What if we represent the given expression in the postfix or Reverse Polish notation:

  5.5 1.1 * 4.5 1.1*+ 7 1.1*+ 8 1.1*+

- The easiest way to implement this is with a stack:

  (5.5 * 1.1) + (4.5 * 1.1) + (7*1.1) + (8*1.1)

Algorithms and Data Structures 1

QUEUE ADT

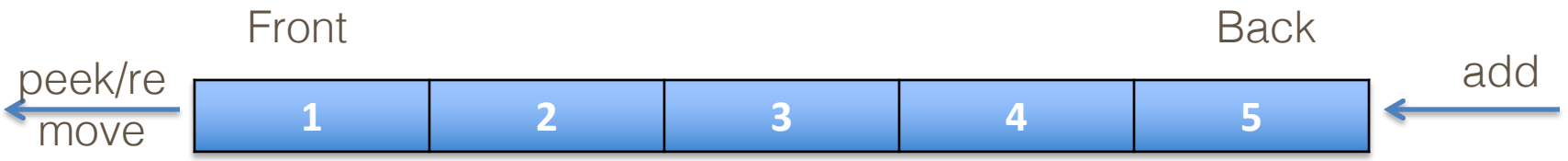
My Least Favorite Queues

[Pictures credit: http://airport.blog.ajc.com, https://s1.cdn.autoevolution.com/images/news/the-longest-traffic-jam-in-history-12-days-62-mile-long-47237-7.jpg]

What is a Queue?

- Queue – a data collection that retrieves elements in the FIFO order (first in, first out)
  - Elements are stored in order of insertion, but don't have indexes
    - Client can only:
      - Add to the end of the queue,
      - Examine/remove the front of the queue

- Basic queue operations:
  - **`Add`** `(enqueue)` - add an element to the back of the queue
  - **`Peek`** - examine the front element
  - **`Remove`** `(dequeue)` - remove the front element



Implementations of Queues

- Like stack, queue can be seen as a list with restriction, and can be implemented as a list:

- Example: `ArrayList` implementation
  - Initially, queue only has elements 1 and 2
  - Add another element, 3, to the queue
  - Add another element, 4, to the queue
  - Add another element, 5, to the queue
  - Remove an element from the queue
  - Remove an element from the queue

What happens when we remove two more elements from the queue?

Implementations of Queues

- What happens when we remove two more elements from the queue?

- Approach – circular array implementation - whenever front or back get to the end of the array, allow them to wrap around to the beginning

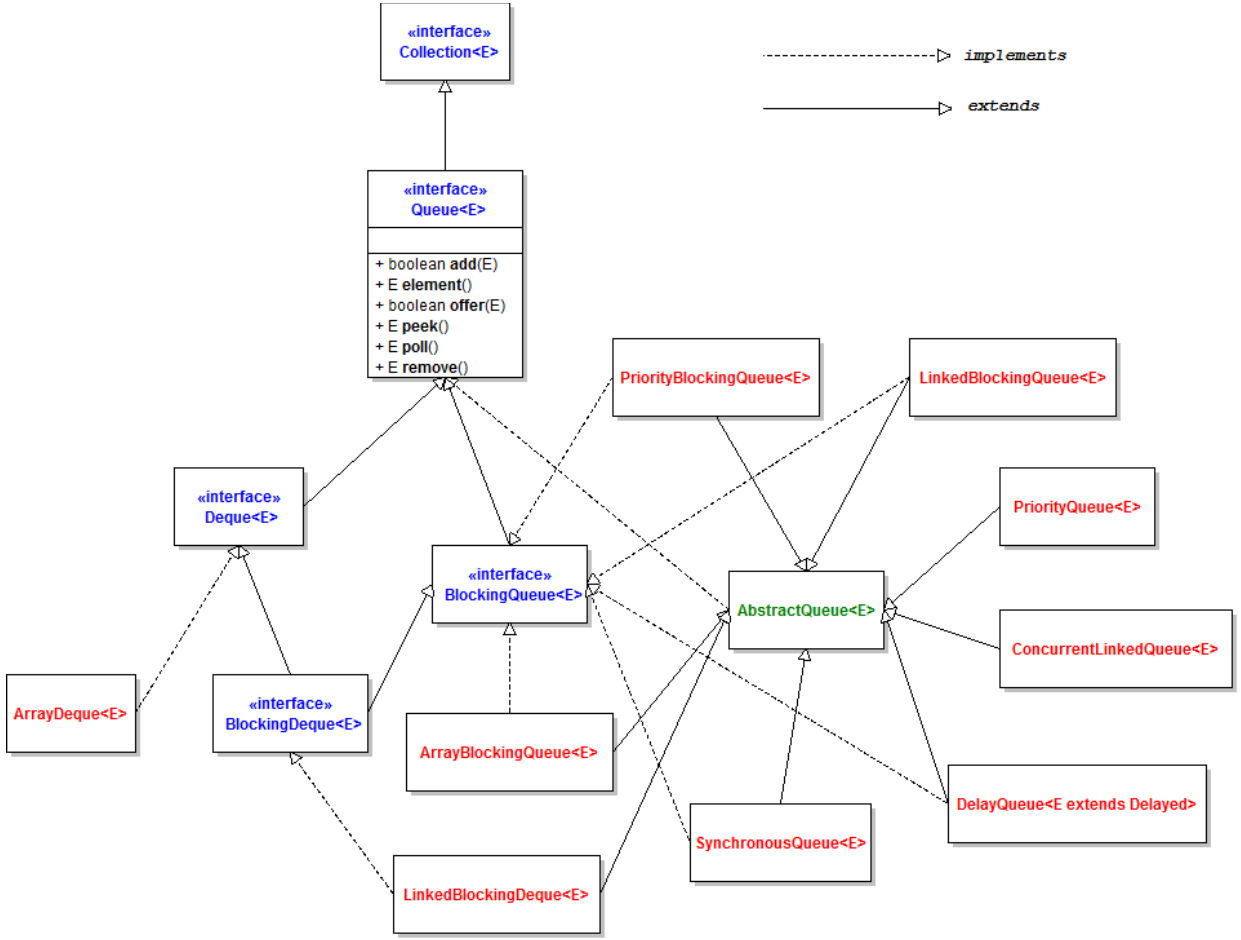
- Example:
  - Add another element, 6, to the queue

  - What is the front and what is the back of the queue now?

Applications of Queues

- Operating systems:
  - Queue of print jobs to send to the printer
  - Queue of programs / processes to be run
  - Queue of network data packets to send
- Programming:
  - Modeling a line of customers or clients
  - Storing a queue of computations to be performed in order
- Real world examples:
  - People waiting in some line
  - ???

Class Diagram of the Queue API



[Pictures credit: http://www.codejava.net/java-core/collections/class-diagram-of-queue-api]

Java Interface `Queue`

`add(value)`	places given value at back of queue
`remove()`	removes value from front of queue and returns it; throws a `NoSuchElementException` if queue is empty
`peek()`	returns front value from queue without removing it; r eturns `null` if queue is empty
`size()`	returns number of elements in queue
`isEmpty()`	returns `true` if queue has no elements

- Example:

```
Queue<Integer> myQueue = new LinkedList<Integer>();
myQueue.add(10);
myQueue.add(16);
myQueue.add(2019); // front [10, 16, 2019] back
System.out.println(myQueue.remove()); // 10
```

Mixing Queues and Stacks

- We often mix stacks and queues to achieve certain effects

- Example: Reverse the order of the elements of a queue

```
Queue<Integer> q = new LinkedList<Integer>();
q.add(1);
q.add(2);
q.add(3); // [1, 2, 3]
Stack<Integer> s = new Stack<Integer>();
while (!q.isEmpty()) {
   s.push(q.remove()); } // Q -> S
while (!s.isEmpty()) {
   q.add(s.pop()); } // S -> Q
System.out.println(q); // [3, 2, 1]
```

Algorithms and Data Structures 1

DEQUE ADT

Deque

[Pictures credit: http://www.java2novice.com/data-structures-in-java/queue/double-ended-queue/]

Deque

	First Element (Head)	
	Throws exception	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`
Remove	`removeFirst()`	`pollFirst()`
Examine	`getFirst()`	`peekFirst()`

	Last Element (Tail)	
	Throws exception	*Special value*
Insert	`addLast(e)`	`offerLast(e)`
Remove	`removeLast()`	`pollLast()`
Examine	`getLast()`	`peekLast()`

[Pictures credit:https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html

Deque

Queue Method	Equivalent Deque Method
`add(e)`	`addLast(e)`
`offer(e)`	`offerLast(e)`
`remove()`	`removeFirst()`
`poll()`	`pollFirst()`
`element()`	`getFirst()`
`peek()`	`peekFirst()`

[Pictures credit:https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html

Deque

Stack Method	Equivalent Deque Method
`push(e)`	`addFirst(e)`
`pop()`	`removeFirst()`
`peek()`	`peekFirst()`

Algorithms and Data Structures 1

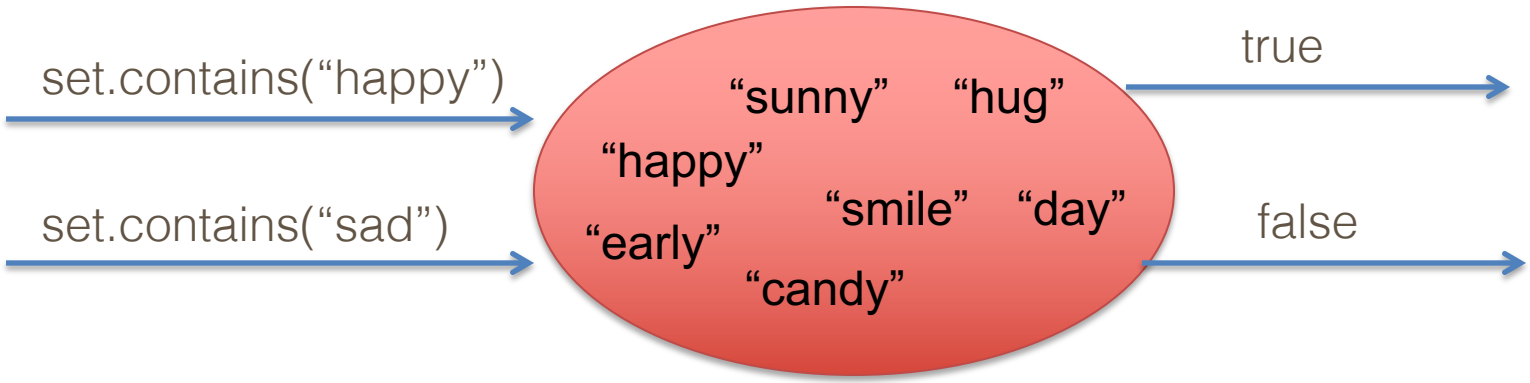
SET ADT

Sets

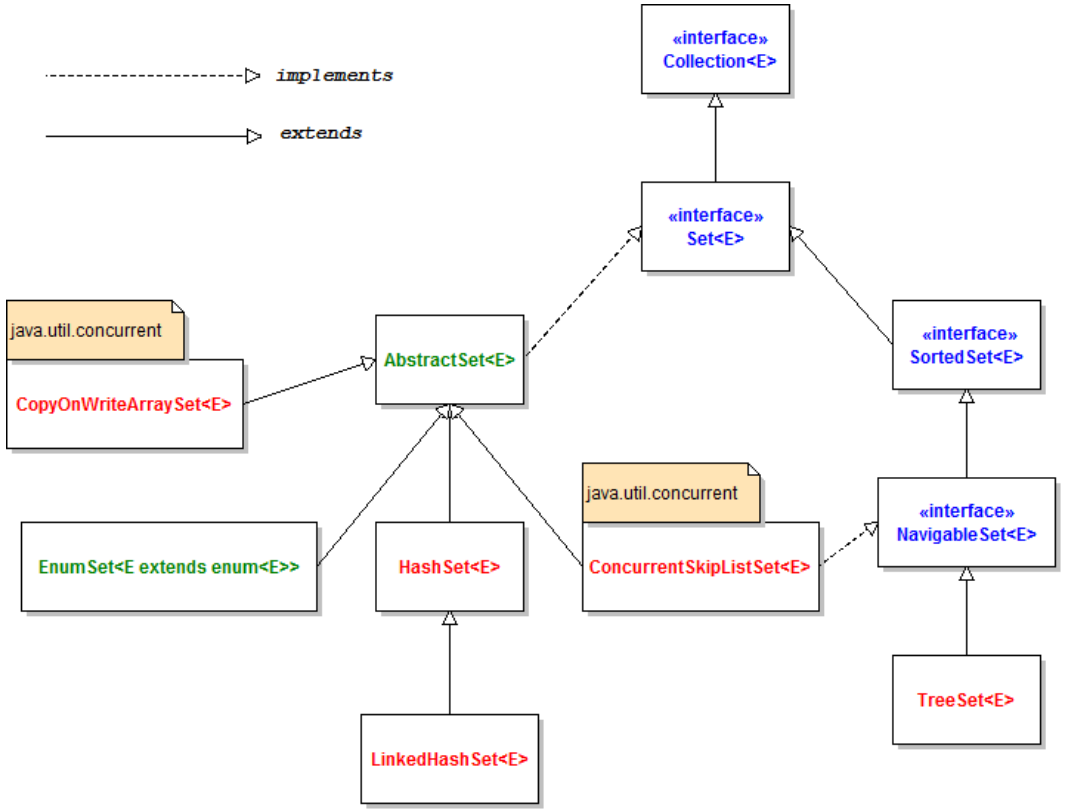
[Pictures credit: https://www.beyondtheblackboard.com/components/com_virtuemart/shop_image/product/full/SET---Box---Transparent-Background---8-22-11_0.png]

Sets

- Set - a collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  - `add`,
  - `remove`,
  - `search (contains)`
- We don't think of a set as having indexes; we just add things to the set in general and don't worry about order



`Set` API Class Diagram



[Pictures credit:
http://www.codejava.net/images/articles/javacore/collections/Set%20API%20class%20diagram.png]

`Set` Implementations

- In Java, sets are represented by `Set` type in `java.util`
- Set is implemented by `HashSet` and `TreeSet` classes
  - `HashSet:` implemented using a "hash table" array
    - Very fast: O(1) for all operations
    - Elements are stored in unpredictable order
  - `TreeSet:` implemented using a binary search tree
    - Pretty fast: O(log N) for all operations
    - Elements are stored in sorted order
  - `LinkedHashSet:`
    - O(1) but stores in order of insertion, but slightly slower than `HashSet` because of extra info stored

`Set` Methods

- We can construct an empty set, or one based on a given collection

- Examples:

```
Set<Integer> set = new TreeSet<Integer>(); // empty

List<String> list = new ArrayList<String>();
...
Set<String> set2 = new HashSet<String>(list);
```

`Set` Methods

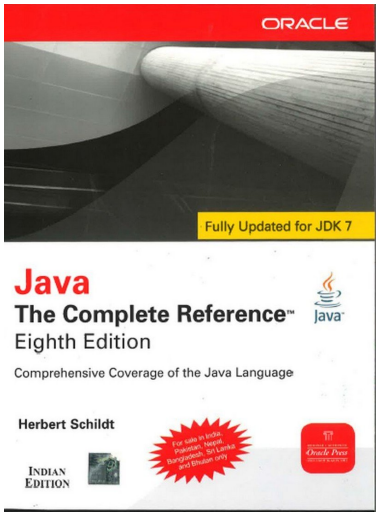
`add(value)`	adds the given value to the set
`contains(`**`value`**`)`	returns true if the given value is found in this set
`remove(`**`value`**`)`	removes the given value from the set
`clear()`	removes all elements of the set
`size()`	returns the number of elements in list
`isEmpty()`	returns true if the set's size is 0
`toString()`	returns a string such as "[3, 42, -7, 15]"

Algorithms and Data Structures 1

MAP ADT

Maps

- Write a program to count the number of occurrences of every unique word in a large text file (e.g. *Java Reference Manual*)



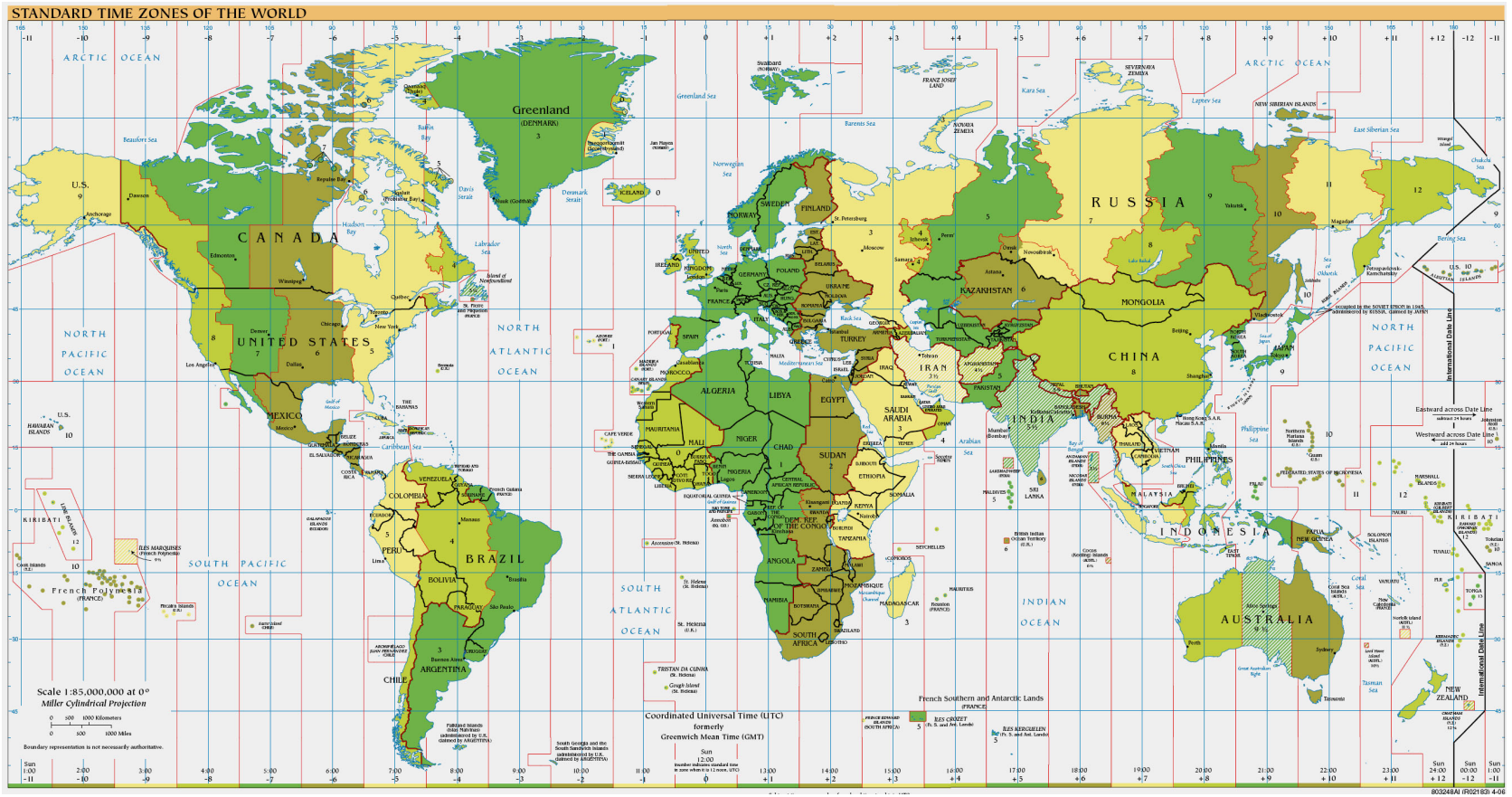
[Pictures credit: https://images-na.ssl-images-amazon.com/images/I/61rKnDmww9L.jpg]

Maps

- Write a program that stores, modifies and retrieves:
  - Assignment grades for every student in this College
  - Financial information for every client of some bank
  - Browsing history for every user of some search engine
  - Searches and transactions for every user of some online retailer
  - Activity and likes of every user of some online platform

- Question: What do these records have in common?
- The way we think about them → every data sample has a unique user → unique ID (key)
- What is the appropriate data collection for this data?
- Maps

Maps



[Pictures credit: https://theodora.com/maps/new9/time_zones_4.jpg]

Maps

- **Map** – a data collection that holds a set of unique *keys* and a collection of *values*, where each key is associated with one value
- Also known as:
  - Dictionary
  - Associative array
  - Hash
- Basic map operations:
  - **`put`**`(key, value)` - adds a mapping from a key to a value
  - **`get`**`(key)` - retrieves the value mapped to the key
  - **`remove`**`(key)` - removes the given key and its mapped value

`Map` Implementations

- In Java, maps are represented by `Map` type in `java.util`
- Map is implemented by the `HashMap` and `TreeMap` classes
  - `HashMap` - implemented using a "hash table"
    - Extremely fast: O(1)
    - Keys are stored in unpredictable order
  - `TreeMap` - implemented as a linked "binary tree" structure
    - Very fast: O(log N)
    - Keys are stored in sorted order
  - `LinkedHashMap` - O(1)
    - Keys are stored in order of insertion

Map Implementations

- Map requires 2 types of parameters:
  - One for keys
  - One for values

- Example:

```
// maps from String keys to Integer values
Map<String, Integer> votes = new HashMap<String, Integer>();
```

`Map` Methods

`put(key, value)`	adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one
`get(`**`key`**`)`	returns the value mapped to the given key (`null` if not found)
`containsKey(`**`key`**`)`	returns `true` if the map contains a mapping for the given key
`remove(`**`key`**`)`	removes any existing mapping for the given key
`clear()`	removes all key/value pairs from the map
`size()`	returns the number of key/value pairs in the map
`isEmpty()`	returns `true` if the map's size is 0
`toString()`	returns a string such as "{a=90, d=60, c=70}"

keySet and Values

- keySet method returns a **Set** of all keys in the map
  - It can loop over the keys in a foreach loop
  - It can get each key's associated value by calling get on the map

Example:

```
Map<String, Integer> ages = new TreeMap<String, Integer>();
ages.put("Marty", 19);
ages.put("Geneva", 2); // ages.keySet() returns Set<String>
ages.put("Vicki", 57);
for (String name : ages.keySet()) {       // Geneva -> 2
    int age = ages.get(name);                 // Marty -> 19
    System.out.println(name + " -> " + age); // Vicki -> 57
}
```

Methods `keySet` and `Values`

- `values` method returns a collection of all values in the map
  - It can loop over the values in a `foreach` loop
  - No easy way to get from a value to its associated key(s)

`keySet()`	returns a set of all keys in the map
`values()`	returns a collection of all values in the map
`putAll(`**map**`)`	adds all key/value pairs from the given map to this map
`equals(`**map**`)`	returns `true` if given map has the same mappings as this one

Example: Maps

- Many words are similar to other words
- For example word wine can become:
  - dine, fine, line, mine, nine, pine, or vine
  - wide, wife, wipe, or wire
  - wind, wing, wink, or wins
- Write a program to find all words that can be changed into at least 15 other words by a single one-character substitution
- Assume that:
  - We have a dictionary consisting of approximately 89,000 different words of varying lengths
  - Most words are between 6 and 11 characters

Algorithms and Data Structures 1

HASHING AND HASH FUNCTIONS

Introduction to Hashing

- Problem: how much does a new phone cost?
  - e.g., Pixel 3??
  - e.g., iPhone XR??

- Ways to answer this question:
  - Look it up in an unsorted list of all phone prices: O(n)
  - Look it up in a sorted list of all phone prices: O(log n)
  - Ask your buddy if she remembers: O(1)

Introduction to Hashing

- Does this difference in time complexity matter?

Num Items	Simple Search $O(n)^2$	Binary Search $O(\log n)$	Buddy $O(1)$
100	10 sec	1 sec	Instant
1000	1.6 min	1 sec	Instant
10000	16.6 min	2 sec	Instant

It sure does!

How do we replicate our buddy with data structures???

Introduction to Hashing

- How do we replicate our buddy with data structures???

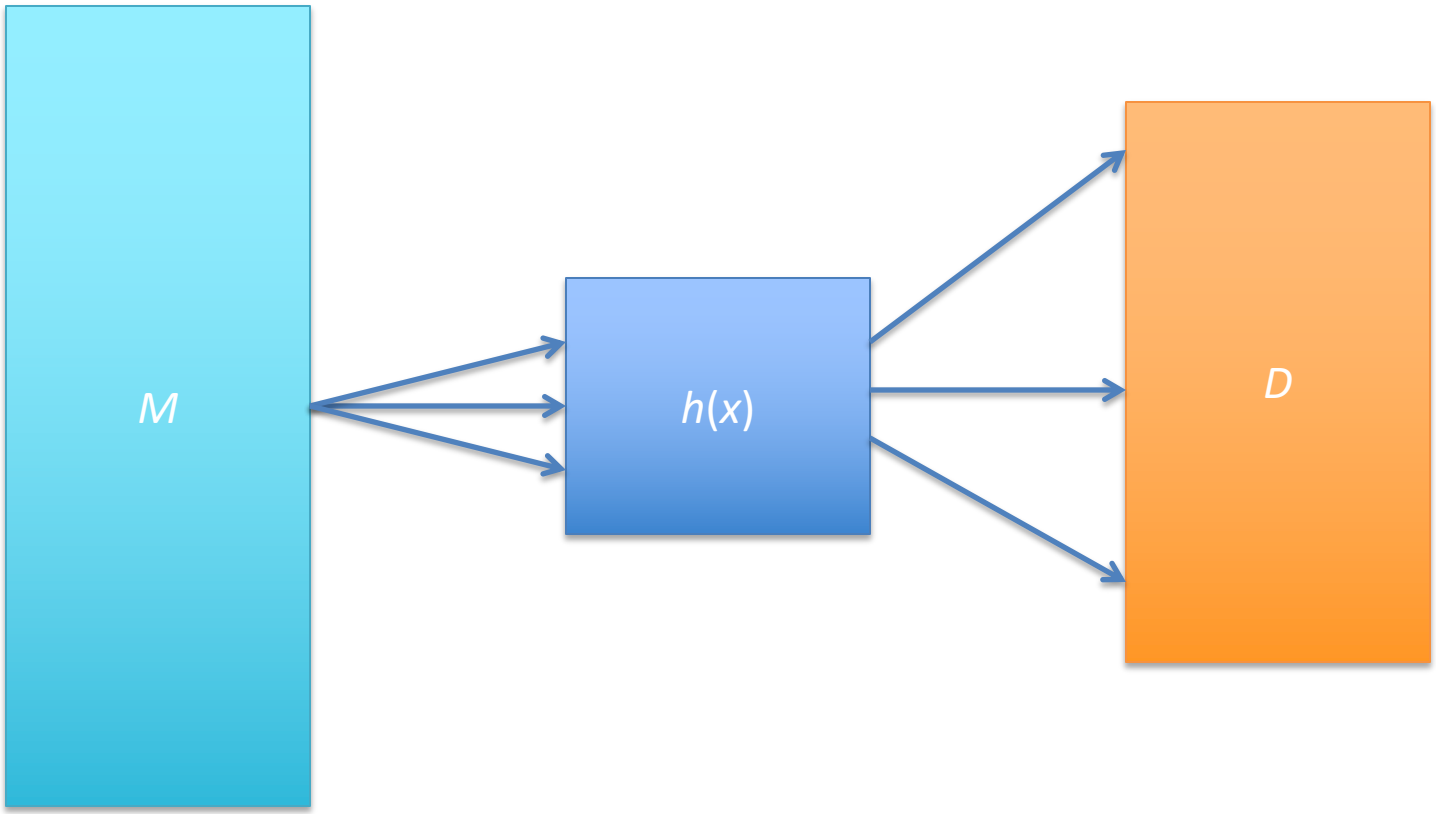
- Observation - each item is essentially 2 items:
    - Product name, and
    - Price
- If we sort by name, we can find a product in O(log n) time
    - How can we get down to O(1)?

Introduction to Hash Functions

- Use a hash function to transform a {string, number, struct, object, . . . } into a number
- Be consistent - every time you give it the same input, it returns the same value
  - Every time you give it “iPhone XR”, it returns ‘xxx’
- Make sure that the output is distinguishing
  - In the best case, it returns a different value for every distinct input given to it
  - Why: a function that always returns 1 is not helpful

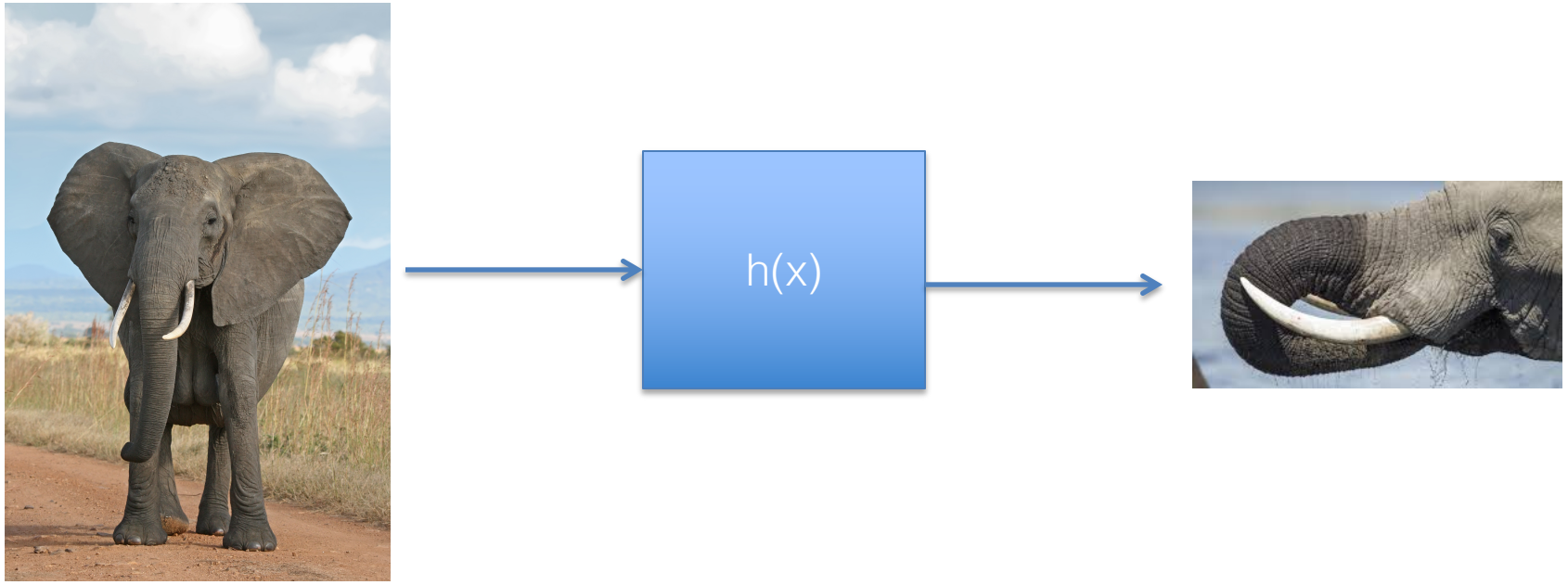
Introduction to Hash Functions

- Another possible approach – pseudo-random mapping using a hash function $h(x)$



Introduction to Hash Functions

- Hash function - maps space M into target space D



[Pictures credit: https://upload.wikimedia.org/wikipedia/commons/3/37/African_Bush_Elephant.jpg https://aos.iacpublishinglabs.com/question/aq/1400px-788px/what-are-elephant-tusks-used-for_3c174bec-bd85-4fab-a617-7a1a33f14c62.jpg?domain=cx.aos.ask.com]

Introduction to Hash Functions

- Hash function - maps the large space M into target space D

- Desired properties of hash functions:
  - Repeatability:
    - For every x in D, it should always be h(x) = h(x)
  - Equally distributed:
    - For some y, z in D, P(h(y)) = P(h(z))
  - Constant-time execution: h(x) = O(1)

Simple Hash Function

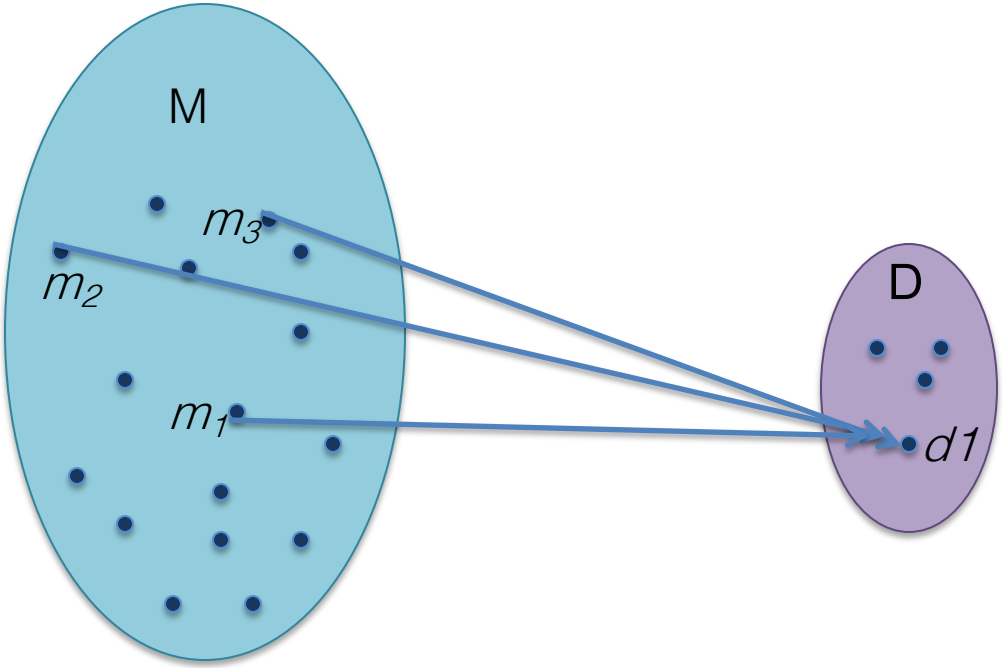
```
/**
* A hash method for String objects.
* @param key the String to hash.
* @param tableSize the size of the hash table.
• @return the hash value.
  */
  public static int hash(String key, int tableSize) {
      int hashVal = 0;

      for(inti=0; i<key.length();i++)
         hashVal = 37 * hashVal + key.charAt(i);

      hashVal %= tableSize;
      if(hashVal < 0)
         hashVal += tableSize;
      return hashVal;
}
```

Problems with Hash Functions

- Hash function – can be thought of as a "lossy compression function", since |M| << |D|



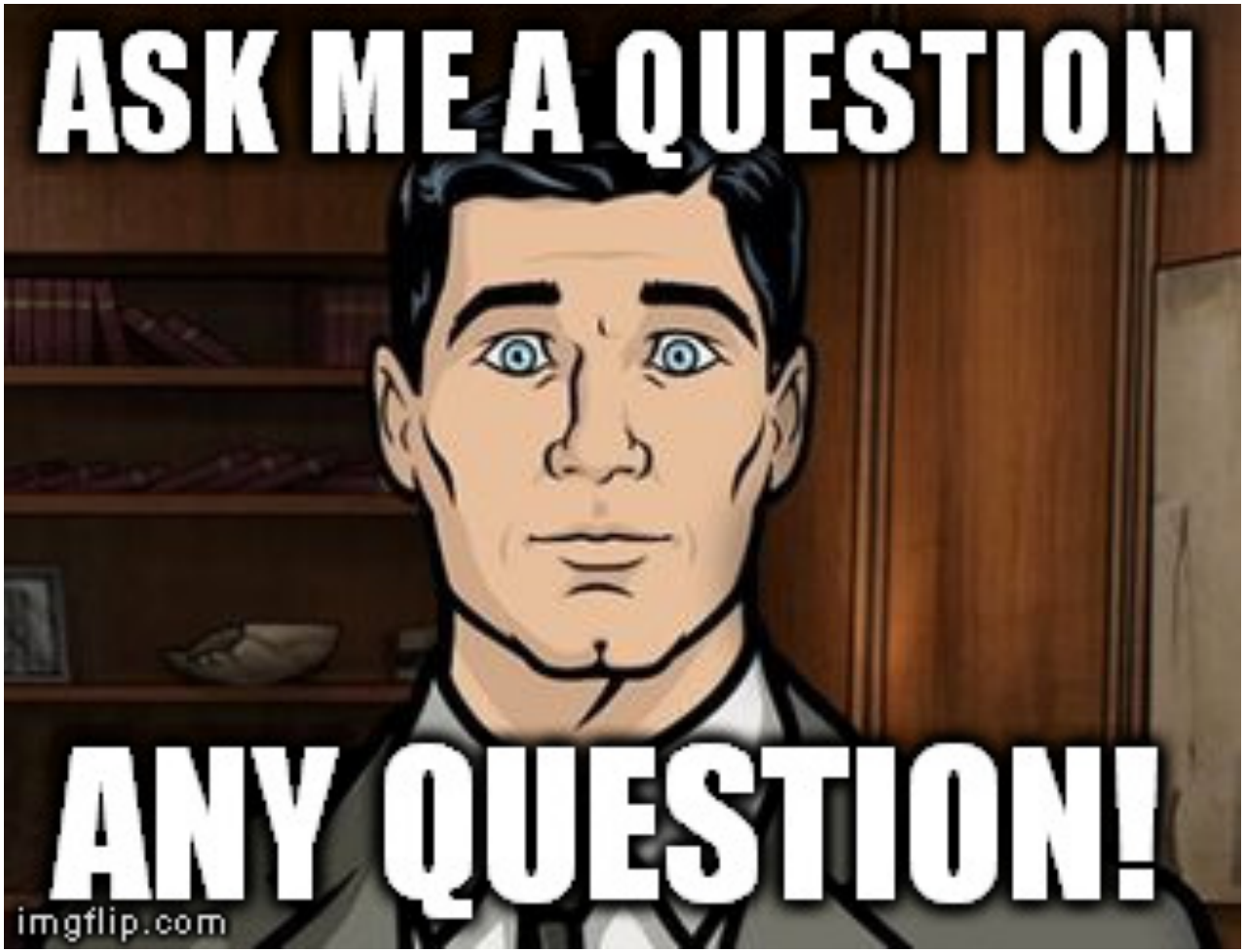
- Do you see any problems here?
- Yes, collision!

Resolving Collisions

- Hash function – can be thought of as a "lossy compression function", since |M| << |D|
- Problem – collision

- Possible approaches to resolve collisions:
  - Store data in the next available space
  - Store both in the same space
  - Try a different hash
  - Resize the array

Your Questions



[Meme credit: imgflip.com]

References and Reading Material

- Mark Allen Weiss, Data Structures and Algorithm Analysis in Java, chapters 1 through 4
- Oracle, java.util Class Collections, [Online] http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html
- Oracle, Java Tutorials Collections, [Online] https://docs.oracle.com/javase/tutorial/collections/
- Vogella, Java Collections – Tutorial, [Online] http://www.vogella.com/tutorials/JavaCollections/article.html
- Oracle, Java Tutorials, Nester Classes, [Online] https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html
- Princetion, Introduction to Programming in Java, Recursion, [Online] http://introcs.cs.princeton.edu/java/23recursion/
- Jeff Ericson, Backtracking, [Online] http://introcs.cs.princeton.edu/java/23recursion/
- Wikibooks, Algorithms/Backtracking, [Online], https://en.wikibooks.org/wiki/Algorithms/Backtracking